BoF: Object Life-Time Handling in Driver Subsystems

Bartosz Golaszewski

Embedded Open Source Summit 2024, Seattle, WA

About me

- Linux kernel developer for the Qualcomm Landing Team at Linaro
- 15 years of embedded linux experience
- Maintainer of the GPIO subsystem
- Author and maintainer of libgpiod
- Open-source contributor to many other projects
- Interested in complex software architecture

Linaro is the software engine of the Arm Ecosystem

Linaro empowers rapid product deployment within the dynamic Arm Ecosystem.

- Our cutting-edge solutions, services and collaborative platforms facilitate the swift development, testing, and delivery of Arm-based innovations, enabling businesses to stay ahead in today's competitive technology landscape.
- Our expertise and contributions spread from Testing & LTS, Security, Cloud & Edge Computing, IoT, AI, CI/CD, Toolchain and Virtualization to vertical projects like Windows on Arm and Android Ecosystem enabling and maintenance.
- Linaro fosters and environment of collaboration, standardization and optimization among businesses and open source ecosystems to accelerate the deployment of Arm-based products and technologies along with representing a pivotal role in open source discovery and adoption.

Linaro has enabled trust, quality and collaboration since 2010

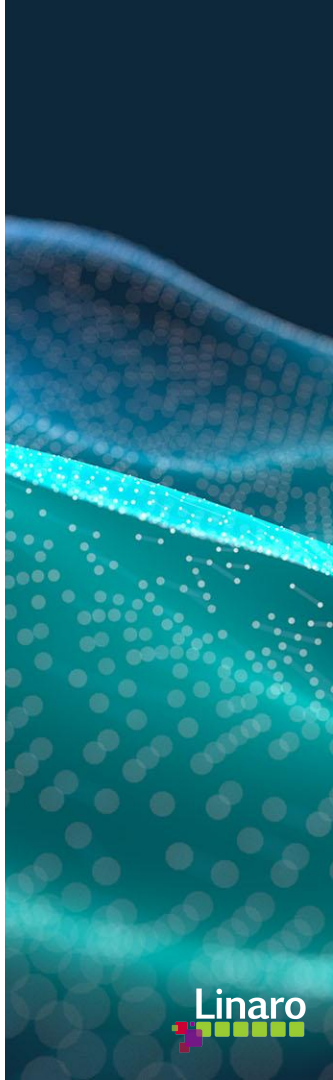
Linaro

Background

- Laurent Pinchart at Linux Plumbers 2022
  - “Why is devm_kzalloc() harmful and what can we do about it”
  - https://www.youtube.com/watch?v=kW8LHWIJPTU
- Bartosz Golaszewski at FOSDEM 2023
  - “Don't blame devres - devm_kzalloc() is not harmful
  - Use-after-free bugs in drivers and what to do about them.”
  - https://archive.fosdem.org/2023/schedule/event/devm_kzalloc/
- Wolfram Sang at EOSS 2023
  - “Subsystems with Object Lifetime Issues (in the Embedded Case)”
  - https://www.youtube.com/watch?v=HCiJL7djGw8
- Bartosz Golaszewski (again) at Linux Plumbers 2023
  - “Improving resource ownership and life-time in linux device drivers”
  - https://www.youtube.com/watch?v=9JZDKBT0nZc
- Wolfram Sang (again again) at Open Source Summit Japan 2023
  - “Object Lifetime Issues - a Threat for Safety?”
  - https://www.youtube.com/watch?v=nnDl89Y7B0k

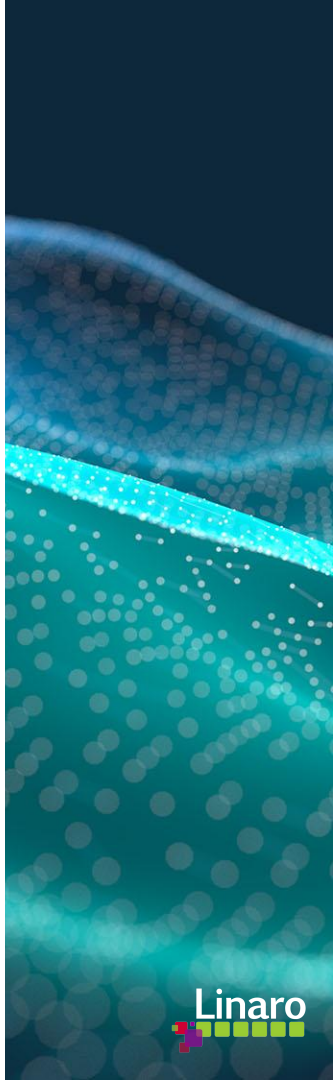
Typical resource provider driver

```
static int foo_probe(struct platform_device *pdev)
{
	struct foo_data *data;
	int ret;

	data = devm_kzalloc(&pdev->dev, sizeof(*data), GFP_KERNEL);
	if (!data)
		return -ENOMEM;

	return devm_bar_subsys_register(&pdev->dev, data);
}
```



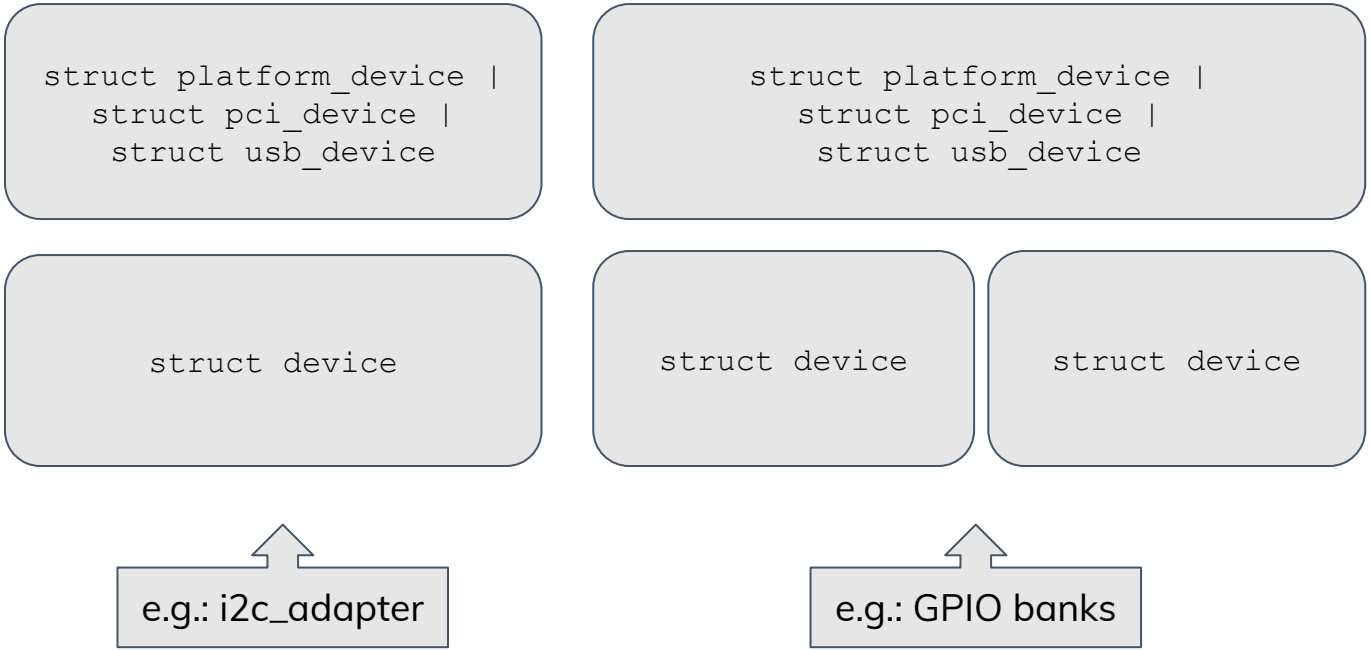
Problem

- Drivers allocate memory for `struct device` in `.probe()` and free it in `.remove()` but `struct device` is reference counted
- Some subsystems just crash, others implement various workarounds
- Solution is easy: `struct device` must survive driver unbind
    - More broadly: the subsystem must be resistant to provider data being destroyed at any moment
- Some subsystems get it right(ish)



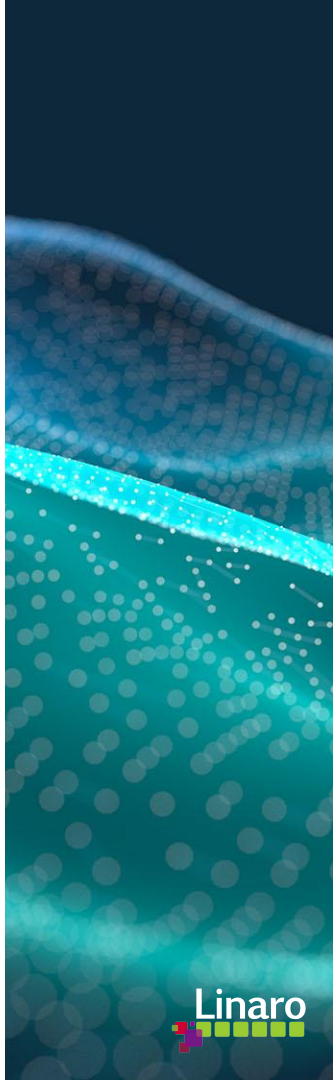
On `struct device`

- Common pattern in device drivers - physical and logical devices:



Agenda

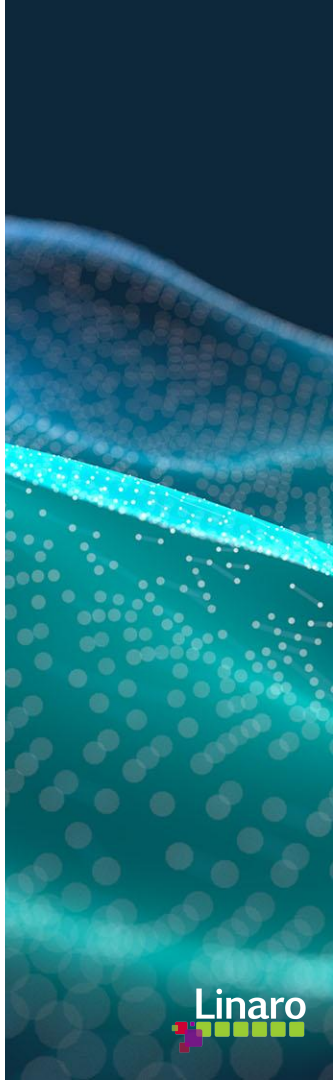
1. **Split responsibility between the subsystem core and provider driver**
2. Allocate driver and subsystem data in the driver and hand it over to the subsystem core
3. Read-only config structure
4. ???
5. Can we rely on device links exclusively?



Split responsibility

- Subsystem defines an “implementation” structure for the provider driver to allocate and register, its life-time is tied to that of the driver

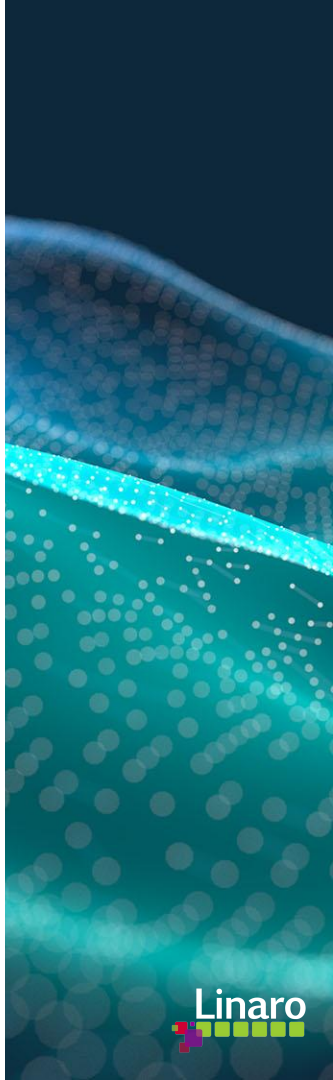
```
struct bar_impl {
    const char *name;
    int (*flob)(struct bar_device *);
};
```



Split responsibility

- Driver “fills” the structure and registers it with the subsystem

```
struct bar_impl *impl;

impl = devm_kzalloc(dev, sizeof(*impl), GFP_KERNEL);
if (!impl)
     return -ENOMEM;

impl->name = "foo";
impl->flob = foo_flob;

return devm_bar_register(dev, impl);
```



Split responsibility

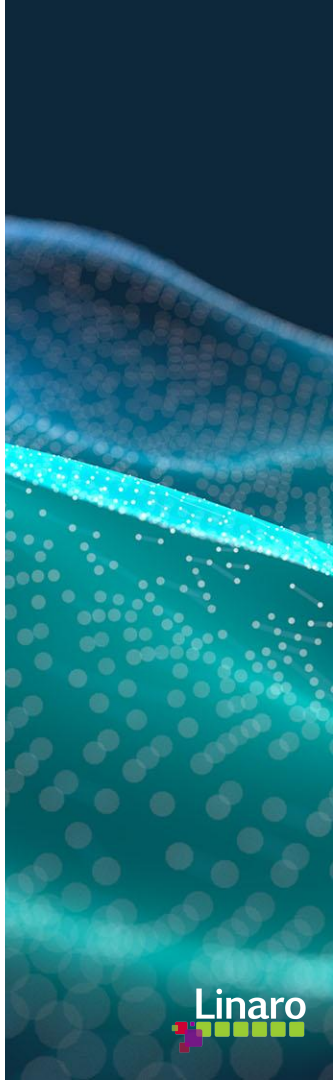
- Subsystem core allocates any additional data inaccessible to the driver, including the reference counted logical `struct device`

```
struct bar_device {
     struct device dev;
     struct bar_impl __rcu *impl;
};

int bar_register(struct bar_impl *impl) {
     struct bar_device *bdev;

     bdev = kzalloc(sizeof(*bdev), GFP_KERNEL);
     if (!bdev)
          return -ENOMEM;

     device_initialize(&bdev->dev);
     bdev->impl = impl;

     return 0;
};
```

Split responsibility

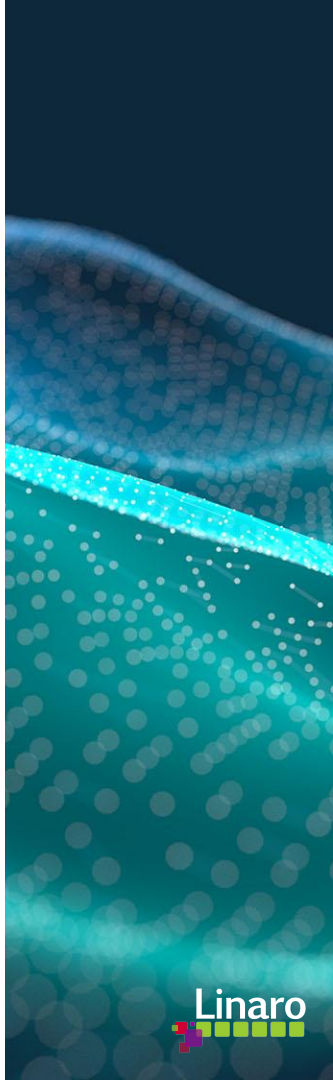
- Subsystem guards the implementation pointer

```
int bar_flob(struct bar_desc *desc)
{
	struct bar_device *bdev = desc->bdev;
	struct bar_impl *impl;

	guard(srcu)(&bdev->srcu);

	impl = srcu_dereference(bdev->impl, &bdev->srcu);
	if (!impl)
		return -ENODEV;

	return bdev->impl->flob(bdev);
}

void bar_unregister(struct bar_device *bdev)
{
	bdev->impl = NULL;
	synchronize_srcu(&bdev->srcu);
}
```



Agenda

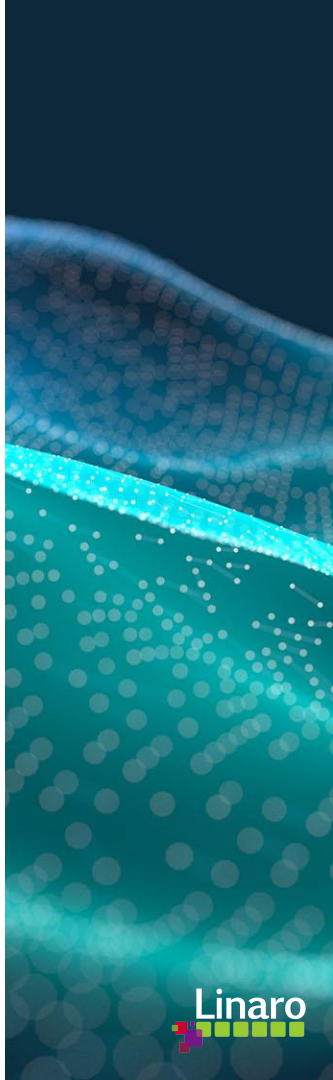
1. Split responsibility between the subsystem core and provider driver
2. **Allocate driver and subsystem data in the driver and hand it over to the subsystem core**
3. Read-only config structure
4. ???
5. Can we rely on device links exclusively?



Handover of subsystem data

- Subsystem defines a structure containing all data required to make the provider driver work

```
struct bar_ops {
    int (*flob)(struct bar_device *);
};

struct bar_data {
    const char *name;
    struct device dev;
    const struct bar_ops *ops;
};
```



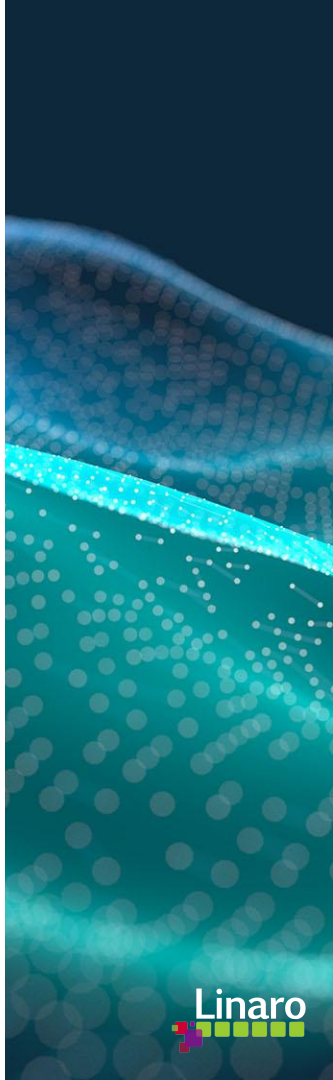
Handover of subsystem data

- Driver allocates the memory for the structure using a non-managed, subsystem-specific interface (and **DOES NOT** free it in `.probe()`) and registers it with the subsystem, effectively handing it over to it.

```
static int foo_probe(struct platform_device *pdev)
{
	struct bar_data *bdata;
	int ret;

	bdata = bar_alloc_data();
	if (!bdata)
		return -ENOMEM;

	ret = devm_bar_register(bdata);
	if (ret) {
		bar_free_data(bdata);
		return ret;
	}

	return 0;
}
```

Agenda

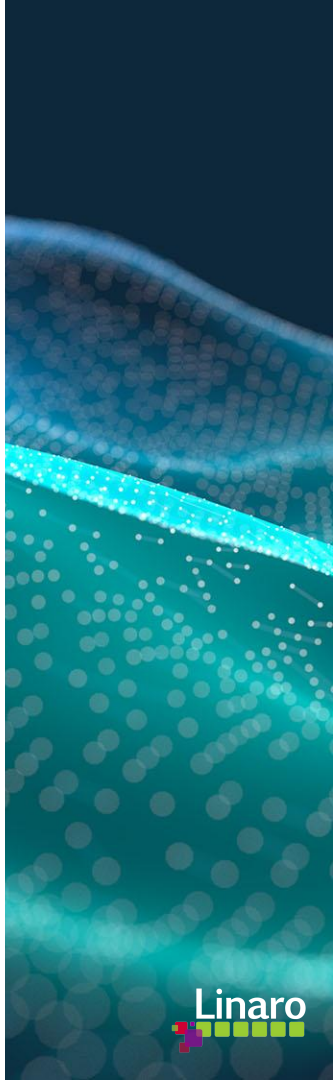
1. Split responsibility between the subsystem core and provider driver
2. Allocate driver and subsystem data in the driver and hand it over to the subsystem core
3. **Read-only config structure**
4. ???
5. Can we rely on device links exclusively?



Read-only config structure

- Variant of the “split responsibility” approach
- Subsystem defines a config structure for the provider driver to provide during registration with the subsystem

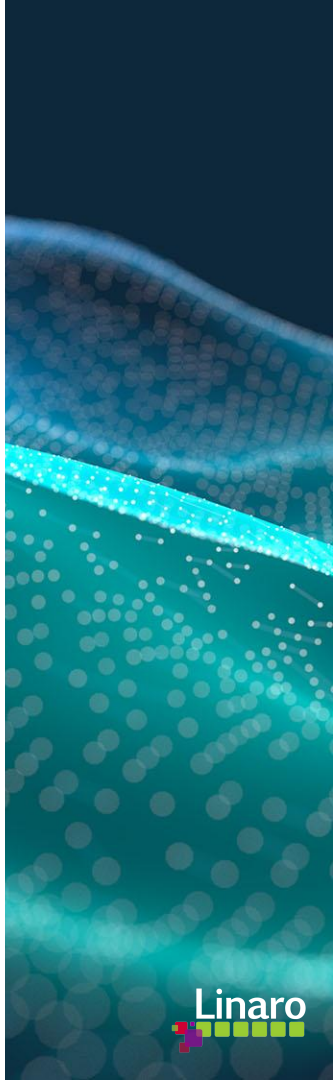
```
struct bar_config {
    const char *name;
    int (*flob)(struct bar_device *);
};
```



Read-only config structure

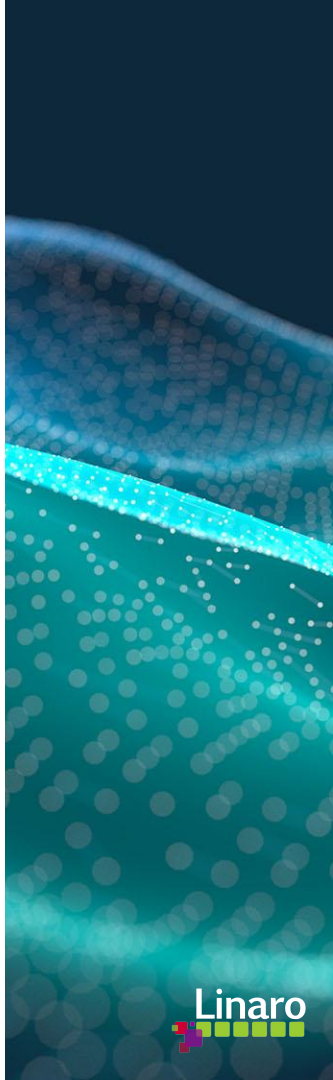
- The structure can be freed after that
- All relevant data is copied and managed by the subsystem

```
struct bar_config config;

memset(&config, 0, sizeof(config))

config.name = "foo";
config.flob = foo_flob;

return devm_bar_register(dev, &config);
```



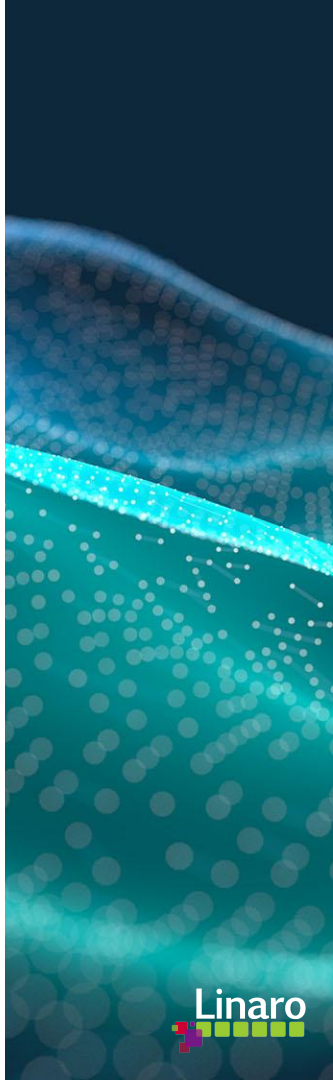
Agenda

1. Split responsibility between the subsystem core and provider driver
2. Allocate driver and subsystem data in the driver and hand it over to the subsystem core
3. Read-only config structure
4. **???**
5. Can we rely on device links exclusively?

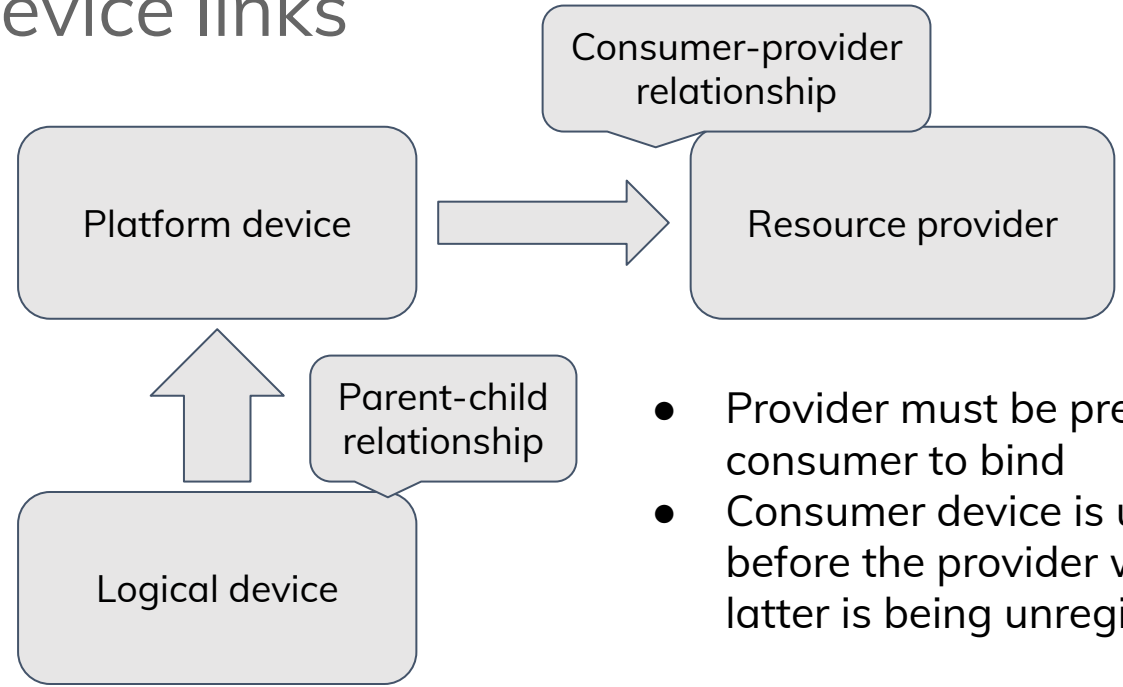


Agenda

1. Split responsibility between the subsystem core and provider driver
2. Allocate driver and subsystem data in the driver and hand it over to the subsystem core
3. Read-only config structure
4. ???
5. **Can we rely on device links exclusively?**



Device links



- Provider must be present for consumer to bind
- Consumer device is unbound before the provider when the latter is being unregistered

Thank you

Visit www.linaro.org
Reach out to me at bartosz.golaszewski@linaro.org